UNIVERSITÉ CÔTE D'AZUR

i3S sophia antipolis

TER Final Report

On the Implementability of Global Types

Supervisors:
Cinzia Di Giusto
Étienne Lozes

Student:
Gabriele Genovese

Academic Year 2024/25
Master in Computer Science / Ubinet Track

Abstract

Behavioral types define how information is exchanged in distributed systems. An example are Multiparty Session Types (MPST), which describe interactions between multiple participants using global protocols and their local counterparts. Ensuring correct implementation, including deadlock freedom and session conformance, is a central concern in MPST. While most research targets peer-to-peer communication, real-world systems often use different communication models such as mailbox-based or causally ordered messaging. A key challenge is that protocols valid in one model may fail in another. In this work, we develop a flexible MPST framework parameterized by different network semantics, including asynchronous, peer-to-peer, causal ordering, and synchronous. We study the implementability problem from a broad semantic perspective, aiming to understand its fundamental limits. My contributions include a survey of related work, a proof of undecidability for weak implementability under synchronous semantics, and enhancements to the ReSCu tool for checking deadlock freedom in synchronous systems. This approach embeds communication models as a parameter, and it provides a basis for verifying distributed systems beyond classical settings.

Contents

1 Introduction

Informally, a *distributed system* is a collection of independent computing entities (interchangeably called processes, actors, nodes, or participants) that communicate and coordinate their actions through message passing over a medium of communication (typically an **asynchronous network**), with the goal of solving a common problem. For example, a client-server application can be seen as a form of distributed system, where the shared objective is to provide services to an end user.

Distributed systems address a wide range of challenges which are difficult to solve without such an architecture, for example reliability under failures safety in critical systems, and consistency of data. Distributed systems are widely adopted in domains such as *cloud computing*, critical infrastructures, and telecommunication-oriented applications (i.e. autonomous cars, aerospace systems, etc.). Given their ubiquity, it is crucial to study every aspect of their **design**, **execution**, and **verification**.

One recurring difficulty is writing **correct programs** in this context. Avoiding programming and logical errors is inherently hard, even for experienced developers. To mitigate this, many abstractions have been introduced, and computer scientists have focused their efforts on developing *formal frameworks* that provide developers with guarantees about their programs. Formal methods for distributed systems offer mathematically rigorous techniques to specify, design, and verify such systems. They are valuable during development, helping detect errors early, and during analysis, enabling the study of critical properties such as **safety**, **liveness**, and **deadlock-freedom**. Two primary verification approaches are *model checking* and *by-construction* verification. Model checking systematically explores a system's state space to confirm properties, while by-construction verification guarantees correctness through the design process itself, preventing errors from being introduced.

There exists several models to reason about distributed systems. Different model are specialized in different aspects of a system, and we are interested in the ones about the exchange of information, such as Calculus of Communicating Systems (CCS), the π-calculus, and Petri nets. In this work, however, we focus on *Multiparty Session Types* (MPST) [18] and *choreographies* [24], since these formalisms place particular emphasis on structured and verifiable communication protocols, making them especially well suited for protocol design. In MPST, communication is specified by a *global type*, which describes the entire interaction among participants. This global type is then *projected* into *local types*, one for each participant. Local types serve as contracts that guarantee each component is compliant to the described protocol, therefore ensuring certain properties, such as deadlock-freedom, at compile time. The implementability problem in MPST is comparable to verifying whether a given global type can be correctly projected into local types, preserving the intended behavior.

1.1 Goal

The goal of this work is to study the **implementability problem**, which concerns whether a global specification can be faithfully realized by a set of *local processes* in a distributed system. In essence, it asks: does an implementation really **respect** the behavior described by a given specification model?

To illustrate the relevance of this problem, consider the following example.

Example 1. Given four processes A, B, C, D distributed over a network, and four messages x, y, z, w to be exchanged according to the description in Listing 1, is it possible to implement it in a real world system?

```
A sends B either message x or y.

If A sends B message x,
    then C sends D message z.

If A sends B message y,
then C sends D message w.
```

Listing 1: Example specification of message exchanges

While the specification can be expressed using several of the formalisms mentioned earlier, only some are capable of revealing that it is, in fact, impossible to implement in a real distributed system. The reason is that process C cannot determine which message to send to D without knowing which message A sent to B, because this information is not locally available to C.

This problem is examined from a theoretical perspective to provide a more formal and precise understanding of the fundamental limits that exist and why syntactical constrains of certain models works. In this work, we use an *automata-based* approach to Global Types. This formalism is designed to be highly modular, incorporating various *network semantics* (such as asynchronous, peer-to-peer, causal ordering, and synchronous semantics) as explicit parameters of the framework. This parameterization allows flexible analysis of different communication models within a unified setting.

The main contributions of this work are:

- a **state-of-the-art** review, highlighting existing research and results in this particular domain and giving perspective to our work;
- we prove **undecidability** of the *weak implementability* problem under the synchronous semantics of our framework;
- we extend and improve the model-checking tool ReSCu [15], enabling verification of *deadlock-freedom* and *progress* for synchronous systems.

The report begins in Section 2 with a state-of-the-art overview, presented in a general and accessible manner, avoiding formal definitions and proofs. Sections 3 and 4 then introduce the necessary formal definitions, followed by the main theoretical contribution and in Section 5 the practical contributions of this work. Finally, Section 6 presents concluding remarks and outlines possible directions for future research and development.

2 State of the art

2.1 Message Sequence Charts

Message Sequence Charts (MSCs) are a standardized graphical formalism, introduced in 1992 [19], used to describe trace languages for specifying communication behavior. Thanks to their simplicity and intuitive semantics, MSCs have been widely adopted in industry. Figure 1 illustrates a simple example based on a minimal client–server architecture. An extension of this formalism, known as High-Level Message Sequence Charts (HMSCs), was later introduced [20]. HMSCs enable the definition of MSCs as nodes connected by transitions and are used to model more complex patterns of message flows by capturing sequences, alternatives, or iterations of atomic MSC scenarios.

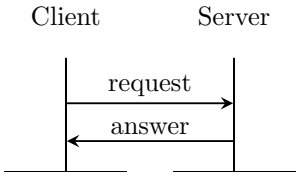


Figure 1: Simple example of a client-server architecture.

The *realizability problem* was first introduced for MSC languages in [1, 2]. It asks whether there exists a distributed implementation that can realize all behaviors of a finite set of MSCs without introducing additional ones. A stronger variant, called *safe realizability*, requires the implementation to also be **deadlock-free**. For finite sets of MSCs, weakly realizability is `coNP`-complete and safe realizability is shown to be decidable in P-time [3]. The problem was subsequently studied for infinite MSC languages, defined as MSC-Graphs (MSGs). For bounded MSGs, safe realizability remains decidable, but weak realizability is undecidable [3]. Extensions of these results to non-FIFO semantics were investigated in [25], corresponding to bag semantics under peer-to-peer communication. Later, Lohrey proved that in the general case, safe realizability is undecidable [23], though it is decidable (and EXPSPACE-complete) for globally cooperative MSGs. Most positive results assume bounded channels, but [5] introduces a new class of HMSCs that allows unbounded channels while maintaining implementability.

2.2 Multiparty Session Types

Multiparty Session Types (MPST) [18] provide a type-theoretic framework to specify and verify communication protocols among multiple participants. They ensure that communication follows a predefined structure, preventing er-

rors such as deadlocks, orphan messages, and unspecified receptions. The **global specification** describes the overall communication protocol. From this, one derives the **local behaviors** of each participant via a *projection* operation. The system's **processes** form the *implementation*, defining how participants interact. With the definition of a *typing system* and suitable *type-checking rules*, one ensures that the implementation conforms to the local specification, thereby guaranteeing properties such as *well-formedness*.

2.2.1 Projectability

A central notion in MPST is *projectability*, which asks whether a global type can be faithfully projected into local specifications for each participant. If projection succeeds, the resulting local types interact without mismatches or unintended behaviors, effectively bridging global specifications and distributed implementations [18]. Projection algorithms, however, often reject natural protocols that fail to meet restrictive syntactic conditions. This tension between expressivity and safety has motivated extensions of the theory, with [8] being the only algorithm aiming for full completeness.

Recent work has focused on strengthening the connection between MPST and automata-theoretic formalisms. Stutz and Zufferey showed that implementability is decidable by encoding global types into HMSCs that are globally cooperative [29, 26]. Building on this, Li et al. [22] proposed a complete projection function for MPST, guaranteeing that every implementable global type admits a correct distributed implementation.

2.2.2 Mixed and Sender driven choice

A key restriction appears in branching. In the original framework [18, 7], choice is **sender-driven**: the first sender dictates the branch, ensuring safety but excluding many common patterns where multiple participants influence the decision [7]. Allowing **mixed choice** increases expressivity by permitting several initiators, but it also makes the implementability problem undecidable in general [4].

Generalized projection operators Stutz's thesis [27] connects MPST to High-level MSCs (HMSCs), introducing a generalized projection operator for sender-driven choice where a sender may branch towards different receivers. This captures patterns beyond classical MPST projection. He also proves that while syntactic projection is incomplete, an automata-theoretic encoding into HMSCs yields decidability for sender-driven choice, with implementability shown to be in `PSPACE`-the first precise complexity bound for this fragment.

Protocol State Machines Later, [28] proposed *Protocol State Machines* (PSMs), an automata-based formalism subsuming both MPST and HMSCs. PSMs show that many syntactic restrictions of global types are not true expressivity limits. Yet, the implementability problem for PSMs with unrestricted

mixed choice remains undecidable, resolving the open question that mixed-choice global types are undecidable in general.

In summary, projectability is well understood for sender-driven choice, where decidability and complexity bounds are established, but moving towards mixed choice inevitably leads to undecidability. Automata-based techniques such as HMSCs and PSMs provide the most powerful tools for extending the theory while preserving decidability in restricted cases.

2.3 Reduction to synchronous semantic

The main idea of this work is that reasoning about implementability becomes more tractable under *synchronous* semantics for automata-based solutions to the implementability problem. In synchronous communication, send and receive actions are tightly coupled, effectively removing nondeterminism caused by asynchronous message buffering. Several results exploit this observation by reducing the implementability problem under richer communication models (e.g. asynchronous or peer-to-peer FIFO) to the simpler synchronous case [3, 11].

Formally, one can show that if a global type is implementable in synchronous semantics, then under certain conditions it is also implementable in more general models such as peer-to-peer or mailbox semantics. This reduction requires constraints such as *orphan-freedom* (no message is left unmatched) and *deadlock-freedom*. The following theorem, currently a work in progress by my supervisors [13], provides a characterization of a connection between peer-to-peer semantics and synchronous semantics:

Theorem 1. *A global type G is implementable in* `p2p` *iff:*

1. $L_{p2p}(proj(G))$ *is a set of sync MSCs;*
2. $proj(G)$ *is orphan-free in p2p;*
3. $L_{p2p}(proj(G))$ *is deadlock-free;*
4. G *is implementable in sync.*

This result highlights the central role of synchronous semantics as a *core model*: implementability in peer-to-peer systems can be reduced to the synchronous case provided the additional safety conditions are met. My own contribution focuses on the last item of the theorem, namely the problem of checking whether a given global type is implementable in synchronous semantics. This question is at the heart of the undecidability results presented in Section 4, and motivates the need for the identification of new subclasses that enable new verification techniques along with tool support.

3 Preliminaries

In this section, the fundamental concepts and definitions necessary to contextualize the main contributions of this work are presented. I, first, introduce

Message Sequence Charts (MSC), followed by an examination of communication models that are particularly interesting. Then, the notions of Global Type and Realizability are defined within the scope of this work, along with the foundational elements required to understand the theoretical contributions.

Definition 3.1 (Message Sequence Chart)**.** Let $\mathbb{P}$ be a finite set of processes and $\mathbb{M}$ a set of messages. An MSC over $(\mathbb{P}, \mathbb{M})$ is a tuple $M = (\mathcal{E}, \rightarrow, \lhd, \lambda)$ where: $\mathcal{E}$ is a finite (possibly empty) set of *events*; $\lambda : \mathcal{E} \rightarrow \Sigma$ is a *labelling function* assigning an action to each event; $\rightarrow$ and $\lhd$ are binary relations on $\mathcal{E}$ satisfying the conditions below. The projection of an MSC M onto a process $i \in \mathbb{P}$ is denoted by $M|_i$. For each process $p \in \mathbb{P}$, define $\mathcal{E}_p = \{\, e \in \mathcal{E} \mid \lambda(e) \in \Sigma_p \,\}$ as the set of events executed by p.

1. **Process relation.** The relation $\rightarrow \subseteq \mathcal{E} \times \mathcal{E}$ relates an event to its immediate successor on the same process: $\rightarrow = \bigcup_{p \in \mathbb{P}} \rightarrow_p$ where $\rightarrow_p \subseteq \mathcal{E}_p \times \mathcal{E}_p$ is the direct successor relation of a total order on $\mathcal{E}_p$.

2. **Message relation.** The relation $\lhd \subseteq \mathcal{E} \times \mathcal{E}$ relates matching send/receive events, satisfying:

   - For every $(e, f) \in \lhd$, there exist processes $p, q \in \mathbb{P}$ and a message $m \in \mathbb{M}$ such that $\lambda(e) = \text{send}(p, q, m)$ and $\lambda(f) = \text{rec}(p, q, m)$.
   - For every receive event f with $\lambda(f) = \text{rec}(p, q, m)$, there exists exactly one $e \in \mathcal{E}$ such that $e \lhd f$.
   - For every send event e with $\lambda(e) = \text{send}(p, q, m)$, there exists at most one $f \in \mathcal{E}$ such that $e \lhd f$.

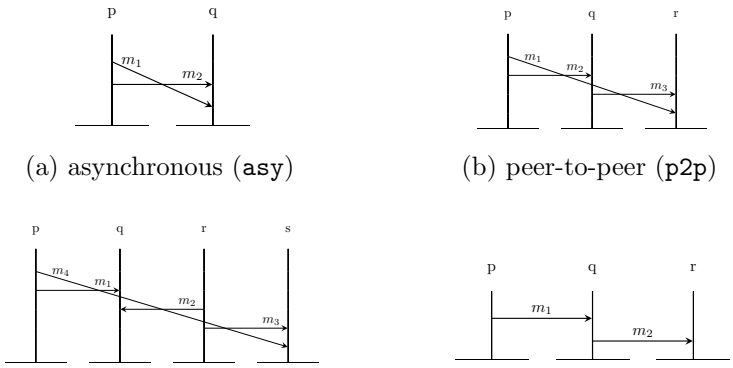
3. **Happens-before relation.** The *happens-before* relation is defined as $\leq_{\text{hb}} = (\rightarrow \cup \lhd)^*$ and is required to be a partial order on $\mathcal{E}$.

3.1 Communication Models

With MSCs, [11] presents some interesting communication semantics. I will describe a few of them informally, using examples to highlight the differences from the main semantics considered in this work, which is `synch`, that is also the only one formally defined in Definition 3.4. Some examples are shown in Figure 2. These examples can be verified that are really part of the corresponding communication semantics with an online tool for MSC [14].

First, we introduce the definition of a linearization of an MSC. A linearization represents a possible ordering of the events in the distributed system, that is, a way to schedule the events of an MSC.

Definition 3.2 (Linearization of a MSC)**.** Let $M = (\mathcal{E}, \Rightarrow, \lhd, \lambda)$ be an MSC. A *linearization* of M is a (reflexive) total order $\rightsquigarrow \subseteq \mathcal{E} \times \mathcal{E}$ such that $\leq_{hb} \subseteq \rightsquigarrow$.



(a) asynchronous (`asy`) (b) peer-to-peer (`p2p`)

(c) mailbox (`mb`) (d) synchronous (`synch`)

Figure 2: MSCs' Examples for various communication models.

Fully asynchronous In the fully asynchronous communication model (`asy`), messages can be received at any time after they have been sent, and send events are non-blocking. This model can be viewed as an unordered "bag" in which all messages are stored and retrieved by processes when needed. It is also referred to as *non-FIFO*. The formal definition coincides with that of an MSC. Figure 2.a illustrates an example of asynchronous communication.

Peer-to-peer In the peer-to-peer (`p2p`) communication model, any two messages sent from one process to another are always received in the same order as they are sent. Alternative names are FIFO. An example is shown in Figure 2.b.

Causally ordered In the causally ordered (`co`) communication model, messages are delivered to a process in accordance with the causal dependencies of their emissions. In other words, if there are two messages m_1 and m_2 with the same recipient, such that there exists a causal path from m_1 to m_2, then m_1 must be received before m_2. This notion of causal ordering was first introduced by Lamport under the name "happened-before" relation. In Figure 2.b, this causality is violated: m_1 should be received before m_3. Causal delivery is commonly implemented using Lamport's logical clock algorithm [21].

Mailbox In this model, any two messages sent to the same process, regardless of the sender, must be received in the same order as they are sent. If a process receives m_1 before m_2, then m_1 must have been sent before m_2. `mb` coordinates all the senders of a single receiver. This model is also called FIFO $n-1$. In Figure 2.c, an example for this communication model is shown.

Synchronous The synchronous (`synch`) communication model imposes the existence of a scheduling such that any send event is immediately followed by its corresponding receive event. An example for this communication model is shown in Figure 2.d.

Definition 3.3 (com-linearisable MSC)**.** An MSC M is *linearisable* in a communication model com if $\mathsf{lin}_{\mathsf{com}}(M) \neq \emptyset$. We write $\mathcal{M}_{\mathsf{com}}$ for the set of all MSCs linearisable in com.

Finally, let's formally define what is a `synch`-MSCs in this context.

Definition 3.4 (Synchronous execution)**.** An execution $e = (w, \mathsf{src}) \in \mathcal{E}_{\mathsf{synch}}$ if for all send event $s \in \mathsf{events}_S(e)$, $s+1$ is a receive event of e and $\mathsf{src}(s+1) = s$.

In other words, an MSC M belongs to $\mathcal{M}_{\mathsf{synch}}$ if all send events are immediately followed by their corresponding receive events.

3.2 Basic notions for Global Types

This part will further highlight the basic notion to understand the formal proof for the theorem presented in Section 4 and, in particular, Global Type and Weakly-realizable. We begin by extending the definition of linearisability so that it applies to all communication models. In our setting, Global Types are automata that describe a language of MSCs.

Definition 3.5 (Global Type)**.** An *arrow* is a triple $(p, q, m) \in \mathbb{P} \times \mathbb{P} \times \mathbb{M}$ with $p \neq q$; we often write $p \xrightarrow{m} q$ instead of (p, q, m), and write Arr to denote the finite set of arrows. A Global Type G is a deterministic finite state automaton over the alphabet Arr.

Example 2. An example of a Global Type expressed as an automaton is the following. Consider the not-implementable specification stated in Listing 1. The protocol can be modelled with the Global Type in Figure 3.

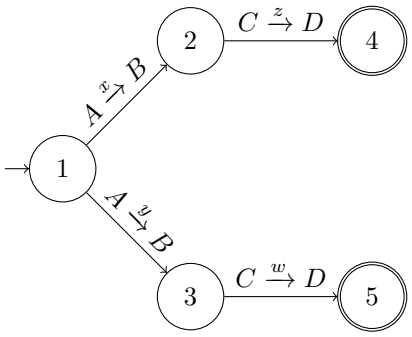


Figure 3: A Global Type, which respects the specification given in Listing 1.

I can now formally define the relationship between MSCs and Global Types. Intuitively, Global Types represent a set of MSCs, allowing us to reason about multiple message sequence scenarios. To obtain this result, let's define what is the existential MSC language $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$ of a Global Type. Let $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$ be the set of sequences of arrows w accepted by G. Informally, the existential MSC language $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$ of a Global Type G is the set of MSCs that admit at least one representation as a sequence of arrows in $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$.

Definition 3.6 ($\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$)**.**

$$\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G}) \stackrel{\mathsf{def}}{=} \{\mathsf{msc}(w) \mid w \in \mathcal{L}_{\mathsf{words}}(\mathsf{G})\}$$

There is also the universal MSC language $\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathsf{G})$ of a Global Type G, defined as the set of MSCs whose arrow-sequence representations all belong to $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$. However, a formal definition is not necessary here.

I now introduce the definition of implementability following the one given by Alur, et al. [3], referred to as *Weak-realizability*. To formalize it, we first define the notions of *weak implication* and *weak closure*.

Definition 3.7 (Weakly-imply)**.** Let $\mathcal{M}$ be a set of MSCs and M another MSC. $\mathcal{M}$ *weakly implies* M, if for any sequence of automata $\langle A_i \mid 1 \leq i \leq n \rangle$, if every MSC in $\mathcal{M}$ is in $L(\prod_i A_i)$ then so is M in $L(\prod_i A_i)$.

Definition 3.8 (Weakly-closure $\mathcal{M}^w$)**.** The weak-closure $\mathcal{M}^w$ of a set $\mathcal{M}$ of MSCs contains all the MSCs $\mathcal{M}$ weakly implies.

Definition 3.9 (Weakly-realizable)**.** An MSC M is said to be weakly-realizable if the set of MSCs $L(M)$ is weakly realizable. A set of MSCs $\mathcal{M}$ is said to be weakly realizable if $\mathcal{M} = \mathcal{M}^w$.

It is important to note that this definition does not include the property of deadlock-freedom.

We are now ready to present the main contributions of this work.

4 Weak-Realizability is Undecidable for Synch Global Types

As already mentioned, the first contribution is Theorem 2, which states that Weak-realizability is undecidable for synchronous global types. To understand this result, I have already covered the basic notions in Section 3 on MSCs, Global Types, and Weak-realizability. These concepts are general and align closely with definitions used in previously established works. We now introduce the main objects employed in the proof of Theorem 2. The proof itself is adapted from the work of Alur et al. [3].

4.1 Definitions

The proof is carried out by reduction from the RPCP problem, which I now recall. The Relaxed Post Correspondence Problem (RPCP) is a variant of the classical Post Correspondence Problem (PCP). RPCP was shown to be undecidable by Alur et al. [3], via reduction from PCP.

Definition 4.1 (Relaxed Post Correspondence Problem)**.** Given a set of tiles $\{(v_1, w_1), (v_2, w_2), ..., (v_r, w_r)\}$, determine whether there exist indices $i_1, ..., i_m$ such that

$$x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m},$$

where $x_{i_j}, y_{i_j} \in \{v_{i_j}, w_{i_j}\}$, such that:

- there exists at least one index i_ℓ for which $x_{i_\ell} \neq y_{i_\ell}$, and
- for all $j \leq m$, $y_{i_1} \cdots y_{i_j}$ is a prefix of $x_{i_1} \cdots x_{i_j}$.

Intuitively, RPCP requires that the concatenation on the left-hand side always grows at least as fast as the right-hand side, while ensuring that at least one chosen tile differs between the two sequences. Moreover, in constructing the strings, we may freely choose which element of each tile (either v_i or w_i) contributes to the left or right sequence.

With this definition in place, I now introduce the main objects used in the proof. Specifically, I begin by showing how a Global Type can represent a single `synch` MSC.

Definition 4.2 (G_M)**.** Given an MSC $M \in \mathcal{M}_{\mathsf{synch}}$, there exists a global type G_M such that $\{M\} = \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$.

We now define a particular Global Type that will be useful in the reduction.

Definition 4.3 (G_S)**.** Given a string $S \in \Sigma^*$, and two integers i, n, the global type G_S is the global type composed of:

- $\mathbb{P} = \{p, q, r, s\}$;
- $\mathbb{M} = \{m_1, m_2, m_3, m_{S_1}, ..., m_{S_c}\}$ where $m_1 = (i, n), m_2 = i, m_3 = (i, n)$, $m_{S_1} = S_1, ..., m_{S_c} = S_c$ with $c = |S|$;
- Arr $= \{p \overset{m_1}{\longleftrightarrow} q \ , \ p \overset{m_2}{\longleftrightarrow} s \ , \ s \overset{m_3}{\longleftrightarrow} r \ , \ q \overset{m_{S_1}}{\longleftrightarrow} r \ , \ ..., \ q \overset{m_{S_c}}{\longleftrightarrow} r \ \}$ where each arrow denotes a synchronous message accompanied by an acknowledgment.

This global type is depicted in Figure 4.

Now, let's generalize Definition 4.2 to include a set of MSCs.

Definition 4.4 (G^*)**.** Given a set of MSCs $\mathcal{M} = \{M \mid \mathcal{M}_{\mathsf{synch}}\}$, G^* is the set of global types such that $G^* = \{G_M \mid M \in \mathcal{M}\}$, where G_M is built using Definition 4.2.

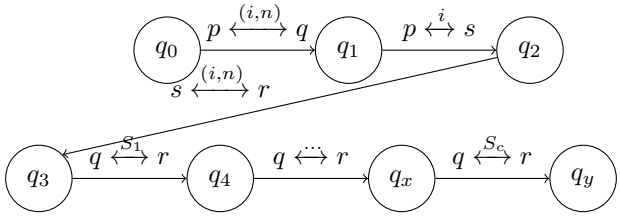


Figure 4: The global type G_S.

Informally, for every MSC $M \in \mathcal{M}$ there exists a global type $G \in G^*$ that captures the language of M.

Definition 4.5 (The L^* global type)**.** Assume a finite set $\mathcal{M}$ of MSCs, where $\mathcal{M} = \{m \mid m \in \mathcal{M}_{\mathsf{synch}}\}$. Let G^* be defined as in Definition 4.4. We define the global type L_N^* as the automaton $\mathcal{A} = (Q, \Sigma, \delta, l_0, F)$ where:

- $Q = \{v_I, v_T\} \cup \bigcup_{G \in G^*} Q^G$;
- $\Sigma = \{\epsilon\} \cup \bigcup_{G \in G^*} \Sigma^G$;
- $\delta : Q \times \Sigma \to 2^Q$ is defined by:
  1. $\forall G \in G^*,\ \delta(v_I, \varepsilon) = q_0^G$ where q_0^G is the initial state of G,
  2. $\forall G \in G^*,\ \forall q_f^G \in F^G,\ \delta(q_f^G, \varepsilon) = v_T$,
  3. $\forall G, G' \in G^*,\ \forall q_f^G \in F^G,\ \delta(q_f^G, \varepsilon) = q_0^{G'}$.
- $l_0 = v_I$ is the initial state;
- $F = v_T$ is the accepting state.

Finally, L^* is obtained by determinizing L_N^*.

Informally, L^* represents all the possible executions of a set of MSCs. To complete the set of definitions, let's define a particular MSC corresponding to Definition 4.3.

Definition 4.6 (M_i^n)**.** For a string u, let u^l denote the l-th character of the string. In the MSC M_i^n, process 1 synchronously sends the label $m_1 = (i, n)$ to process 2, then transmits the index $m_2 = i$ to process 4. Subsequently, process 4 sends $m_3 = (i, n)$ synchronously to process 3. After these control messages, process 2 sends the characters $m_i^1 = x_i^1, ..., m_i^c = x_i^c$ synchronously to process 3 (where c is the length of x_i). This MSC is depicted in Figure 5, where $n \in \{0, 1\}$ and: if $n = 0$, then $x_i = v_i$; if $n = 1$, then $x_i = w_i$.

Finally, let's state the lemmas and main theorem of this work, along with all the proofs.

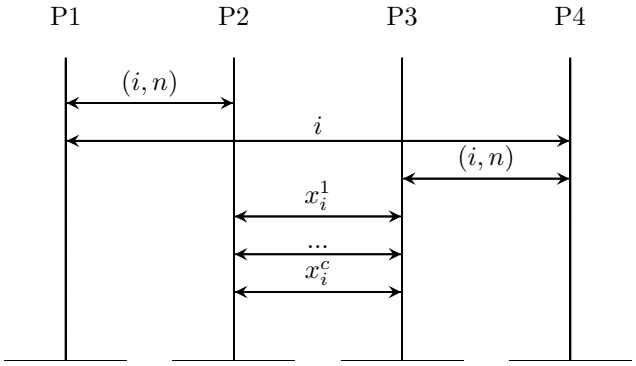


Figure 5: The M_i^n MSC.

4.2 Lemmas and main proof

Before presenting the main undecidability result, we first establish two auxiliary lemmas that characterize the behavior of the MSCs used in the reduction. These lemmas will serve as building blocks for the proof of Theorem 2.

Lemma 1. *The MSC M_i^n belongs to $\mathcal{M}_{\mathsf{synch}}$.*

Proof. By Definition 3.3 and Definition 3.4, we need to show a linearization with all send operations followed by their corresponding receive operations:

$$\{\ !m_1?m_1\ !m_2?m_2\ !m_3?m_3\ !m_i^1?m_i^1 \ldots !m_i^c?m_i^c\ \}.$$

Such a linearization exists by construction, hence M_i^n is synchronous. □

Lemma 2. *The MSC M_i^n (Def. 4.6) is included in $L(G_S)$, where G_S is the global type defined in Def. 4.3.*

Proof. Both M_i^n and G_S describe the same communication structure: process p sends (i, n) to q and i to s; process s relays (i, n) to r; process q then sends the characters of S (here matching x_i) to r. If i, n and S are the same, the sequence of messages is identical in both M_i^n and G_S. Since both models enforce synchronous communication, their linearizations coincide. Hence, $M_i^n \in L(G_S)$. □

Theorem 2. *Given a global type G, checking if G is weakly-realizable is undecidable.*

Proof. The proof proceeds via a reduction from the RPCP problem. Let's define some useful elements for the proof.

Given an instance $\Delta = \{(v_1, w_1), \ldots, (v_m, w_m)\}$ of RPCP, we construct a set L of MSCs over four processes as follows. For each pair (v_i, w_i), we define two MSCs, M_i^0 and M_i^1, as illustrated in Figure 5. Observe that the communication graph of each MSC is strongly connected and involves all four processes. Therefore, the MSC represented from L^* and derived from L is bounded. With the set L, and following the Definition 4.5, we can construct the global type L^*.

We need to prove:

$\Delta \in$ RPCP if and only if the global type L^* is not weakly-realizable.

$\Rightarrow$ Let's assume that $R = (i_1, a_1, b_1, i_2, a_2, b_2, \ldots, i_m, a_m, b_m)$ are the indices for a solution to a generic RPCP problem instance, and the bits a_j and b_j indicate which string (v_{i_j} or w_{i_j}) is chosen to go into the two (left and right) long strings. Consider the new MSCs M and M' obtained from the sequences $M = M_{i_1}^{a_1} \cdots M_{i_m}^{a_m}$ and $M' = M_{i_1}^{b_1} \cdots M_{i_m}^{b_m}$. Executions of both of these (sequences of) MSCs must exist in any realization of L^*. Additionally, these MSCs are in $\mathcal{M}_{\mathsf{synch}}$ because they are sequence of $\mathcal{M}_{\mathsf{synch}}$ MSC (Lemma 1). M corresponds to the construction of the left side of equation (1) of the RPCP problem, and, instead, M' represents the construction of the right side. We then look at the projections $M|_1$, $M|_2$, $M|_3$, and $M|_4$ of M, and $M'|_1$, $M'|_2$, $M'|_3$, $M'|_4$ of M' onto the 4 processes. Now consider an MSC M'' formed from $M'|_1$, $M'|_2$, $M|_3$, and $M|_4$. This MSC represent the construction of the solution to the problem. Processes 1 and 2 construct the right part ($y_{i_1} ... y_{i_m}$) and processes 3 and 4 construct the left part ($x_{i_1} ... x_{i_m}$). The claim is that the combined MSC M'' is weakly implied by L^*. By definition, the only thing to establish is that M'' is indeed an MSC, in the sense that it is acyclic, well-formed, complete and synchronous. The only new situation in terms of communication in M'' is the communication between P_1 and P_4, and between P_2 and P_3. But the communication between P_1 and P_4 is consistent in $M'|_1$ and $M|_4$ (i.e., the sequence of messages sent from P_1 to P_4 in $M'|_1$ is equal to the sequence of messages received in $M|_4$), and the communication between P_2 and P_3 is consistent in $M'|_2$ and $M|_3$ because R is a solution to the RPCP. Furthermore, the acyclicity of M'' follows from the property of the solution that the string formed by the first j words on processes 1 and 2 is always a prefix of the string formed by the first j words on processes 3 and 4. Consequently, each message from P_1 to P_4 is sent before it needs to be received.

Finally, we prove that $M'' \in \mathcal{M}_{\mathsf{synch}}$. Assume, for contradiction, that $M'' \notin \mathcal{M}_{\mathsf{synch}}$. Then, there should be a cycle of dependencies in the communication pattern. There are no communication between P_2 and P_4, and between P_1 and P_3. Therefore, this cycle must involve all processes, starting for example from P_1 and having this dependency graph $P_1 \leftrightarrow P_2 \leftrightarrow P_3 \leftrightarrow P_4 \leftrightarrow P_1$. The only new situation that we now that can cause a cycle are the communication between P_1 and P_4, and between P_2 and P_3. We don't need to analyze the new communication between P_1 and

P_4 because it's not feasible in any communication model, but we need to analyze the one between P_2 and P_3 because it's feasible in FIFO.

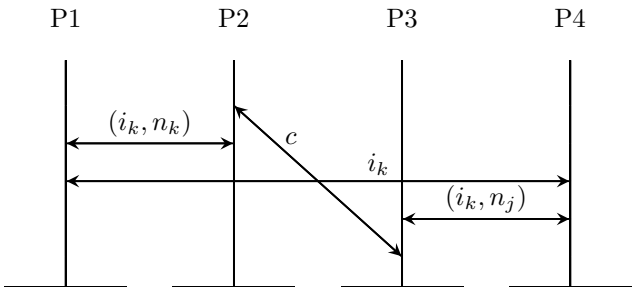


Figure 6: The M_i^n MSC.

For the communication between P_2 and P_3, the only possible cycle pattern is depicted in Fig. 6. Suppose P_2 wants to send a character c, but P_3 is not expecting any further characters. In order for P_3 to resume receiving, it must first receive an index from P_4. However, P_4 can only send this index after receiving it from P_1, which in turn must first communicate the index to P_2. At this point, P_2 needs to receive the index from P_1, but it cannot do so until it finishes sending character c. This creates a circular dependency among the processes, making the communication pattern impossible. This cycle would break the prefix property as $x_1...x_{k-1}...x_m = y_1...y_{k-1}...y_m$, but the character c appears in $y_1...y_{k-1}$ but not in $x_1...y_{k-1}$ contradicting the assumption that $y_1...y_{k-1} \leq x_1...x_{k-1}$. Therefore, we conclude that $M'' \in \mathcal{M}_{\mathsf{synch}}$.

Note that M'' cannot itself be in L^* because there must be some index i_j where $a_j \neq b_j$, and no MSC exists in L where, after P_1 announces the index, what P_2 sends is not identical to what P_3 receives.

$\Leftarrow$ Suppose there is some MSC $M^{@}$ which exists in any realization of L^*, but is not in L^* itself. We want to derive a solution to Δ from $M^{@}$. First, it is clear that the projection $M^{@}|_1$ must consist of a sequence of pairs of messages (the first of each pair acknowledged), sent from process 1 to processes 2 and 4, respectively, with messages (i, b) and i. Likewise, it is clear that, in order for process 2 to receive those messages, $M^{@}|_2$ must consist of a sequence of receipts of (i, b) pairs, and after each (i, b), either v_i or w_i is sent to process 3, based on whether $b = 0$ or $b = 1$, before the next index pair is received. Likewise, $M^{@}|_4$ consists of a sequence of receipts of index i from process 1, followed by sending of $(i, 0)$ or $(i, 1)$ to process 3, and $M^{@}|_3$ consists of a sequence of receipt of $(i, 0)$ or $(i, 1)$ followed by receipt of v_i or w_i, respectively. Now, since $M^{@}$ is not in L^*,

for some index i the choice of v_i or w_i must differ on process 2 and process 3. (Note, we are assuming that the buffers between processes are FIFO.) Furthermore, because of the precedences, the prefix formed by the first j words on process 2 must precede the $(j+1)$-th message from process 1 to process 4, which in turn precedes the $(j+1)$-th message from 4 to 3, and hence the $(j+1)$-th word on process 3. That is, the string formed by the first j words on process 2 is a prefix of the string formed by the first j words on process 3. Therefore, we can readily build a solution for Δ from $M^{@}$ by building the strings of the solution taking the projections of P_1 and P_4. In fact, P_1 builds y and P_4 builds y.

□

The sequence of lemmas and the main theorem collectively establish the undecidability of weak-realizability for global types.

5 ReSCu

I now present ReSCu (first introduced in [9, 12, 16]), describing its features, the input language it uses, its implementation, and the modifications I introduced to extend its functionality [10]. The updated repository with the new examples is available on GitHub [15].

5.1 Characteristics

ReSCu is a command-line tool that can check both membership in the class of `synch` systems (called Realizable with Synchronous Communication or, in brief, RSC from now on) and reachability of regular sets of configurations. It accepts input systems with arbitrary topologies and supports both FIFO and bag buffers. The tool provides several options: `-isrsc` checks whether the system is RSC, and `-mc` checks reachability of bad configurations. Both checks can be combined in a single run. The `-fifo` option overrides buffer types by treating all as FIFO. When a system is unsafe, the `-counter` option (used with `-mc`) produces an RSC execution that leads to the bad configuration, while the same option used with `-isrsc` outputs the borderline violation execution if the system is not RSC. Additional features include a progress display to estimate remaining runtime during long computations, and `-to_dot`, which exports the system to DOT format for visualization. One of the most similar tools is McScm [17], that uses a framework with for different verification techniques. Symbolic Communicating Machines (SCM), defined and used in [6] serve as the input format of the tool. The grammar has been updated to provide greater flexibility and clarity. In particular, transition guards have been made optional (with a default value `: when true`), and a new `final` keyword has been introduced to explicitly specify final states. The updated grammar is shown in Listing 2, found in Appendix 6.1. A formal definition for this object is stated in Definition 5.2.

5.2 Progress and Deadlock-Freedom

I extended ReSCu with verification routines that focus on two fundamental correctness properties of distributed systems: *progress* and *deadlock-freedom*. To enable this, the tool constructs the synchronous system using the synchronous product whenever the input SCM is recognized as an RSC. These two verification routines are triggered only after another check: once the system is proven to be RSC, we can safely construct a well-formed synchronous product from it. Let's first define basic notion to understand this operation.

I assume standard notations for automata, words, and languages.

Definition 5.1 (NFA)**.** A nondeterministic finite automaton (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, l_0, F)$, where Q is the set of states, Σ the alphabet, $\delta : Q \times \Sigma \to 2^Q$ the transition relation, l_0 the initial state, and $F \subseteq Q$ the set of accepting states. We denote by $\mathcal{L}_{\mathsf{words}}(\mathcal{A})$ the language of $\mathcal{A}$. Deterministic automata and ε-transitions are defined in the standard way.

Definition 5.2 (CFSM)**.** A *communicating finite state machine* (CFSM) is an NFA with ε-transitions $\mathcal{A}$ over the alphabet Act. A system of CFSMs is a tuple $\mathcal{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$.

This definition corresponds to the concept of an SCM. We now present the definition of the Synchronous Product, which has been implemented in the tool and serves as a key component for analyzing.

Definition 5.3 (Synchronous Product)**.** Let $\mathcal{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be a system of CFSMs, where $\mathcal{A}_p = (L_p, Act_p, \delta_p, l_{0,p}, F_p)$ is the CFSM associated to process p.

The *synchronous product* of $\mathcal{S}$ is the global type $\mathrm{prod}(\mathcal{S}) = (L, Arr, \delta, l_0, F)$, where

- $L = \prod_{p \in \mathbb{P}} L_p$ is the set of global locations,
- $l_0 = (l_{0,p})_{p \in \mathbb{P}}$ is the initial global state,
- $F = \prod_{p \in \mathbb{P}} F_p$ is the set of global final states,
- δ is the transition relation defined as follows: $(\vec{l},\ p \xrightarrow{m} q,\ \vec{l}\,') \in \delta$ if

$$(l_p,\ !m^{p \to q},\ l'_p) \in \delta_p, \quad (l_q,\ ?m^{p \to q},\ l'_q) \in \delta_q, \quad l'_r = l_r \ \text{ for all } r \notin \{p, q\}.$$

After constructing the synchronous product, the tool performs several important post-processing operations. In particular, it removes any unreachable nodes from the resulting product, simplifying the structure and ensuring that only relevant states are retained for further analysis. We can now define the two properties added to the tool.

I denote Reach as the function that, given a node, gives the set of reachable nodes, and Init, Final as the initial and final node. Informally, a given system satisfies *progress* if, from every reachable node, either the system can eventually perform a transition, or the node is a final one.

Definition 5.4 (Progress)**.** A system S has the progress property iff $\text{Prod}_{sync}(S)$ has the property Reach(Init) $\subseteq$ Reach(Final).

In other words, this property is implemented by checking for all nodes that are not a final state if they are without any outgoing transitions. This definition allows infinite loops. Informally, a system is *deadlock-free* if no reachable non-final node exists in which all processes are blocked (i.e., no further actions can be taken).

Definition 5.5 (Deadlock-Freedom)**.** A system S is deadlock-free iff $\text{Prod}_{\text{sync}}(S)$ has the property Reach(Init) $\subseteq$ Prefixes(Final).

More precisely, a system that can reach, from its initial states, some state that does not lead to a final state is not deadlock-free. Under this definition, even a loop that never reaches a final state is considered a deadlock, making the property more restrictive. This check is implemented using a reverse search algorithm starting from the final states.

Additionally, the synchronized system can be exported in DOT format (with a default filename of `sync.dot`), which allows for graphical visualization of its structure and behavior. Let's finally see some relevant examples.

5.3 Examples

To illustrate these notions, I present two examples. The first is the classical *Dining Philosophers* problem, which shows how resource contention can lead to deadlock. The second is a minimal looping system that demonstrates how a process may satisfy the progress property while still failing to be deadlock-free.

Example 3. Consider two philosophers P_0, P_1 and two forks F_1, F_2, arranged so that each philosopher needs both forks to eat. If both philosophers pick up their left fork simultaneously, each waits indefinitely for the other fork, producing a deadlock. This captures the essence of the Dining Philosophers problem: concurrent processes blocking one another when competing for shared resources.

The behavior of the four participants is shown in Figure 8 of Appendix 6.1. Running the tool on this input produces the terminal output in Listing 3 and the corresponding synchronous system in Figure 7, both included in the appendix. In the generated figure, the red state marks a configuration where no further actions are possible, while the three yellow states correspond to deadlocks, i.e. executions where both philosophers wait for each other indefinitely. The terminal output also lists the precise configurations of these problematic states.

Example 4. Now consider two processes A and B that exchange data. At some point, each makes a nondeterministic choice: one branch continues sending messages indefinitely, while the other leads to termination. Once the choice to continue is taken, however, there is no way to return to the terminating branch. As a result, the system may remain stuck in an infinite loop, never reaching

a final state. Although both processes remain active, the system is effectively deadlocked.

The behavior of this system is shown in Figure 10 of Appendix 6.1. Executing the tool produces the output in Listing 4 and the synchronous system in Figure 9, also included in the appendix. In the generated figure, yellow states highlight the deadlocked executions, while the terminal output provides the configuration of each detected deadlock.

6 Conclusion

This work addressed the *implementability problem* for Global Types, a central concern in the verification of distributed systems. After surveying the state of the art, I positioned our contribution within an ongoing research effort, bridging well-established theoretical foundations with practical tool development.

On the theoretical side, I introduced the necessary background notions-CFSMs, Global Types, MSCs, and communication models-and formalized weak realizability. The main contribution was to connect the implementability problem to classical undecidability results, in particular through a reduction to the Relaxed Post Correspondence Problem (RPCP).

On the practical side, I improved and extended the ReSCu tool, used for checking realizability and other semantic properties of Symbolic Communicating Machines (SCMs). The input grammar was refined for greater usability, and new verification routines were implemented, including checks for progress and deadlock-freedom. The tool now also generates visual representations of synchronous systems, along with illustrative examples. These extensions strengthen ReSCu both as a research prototype and as a practical aid for automated verification.

Overall, the contributions span two complementary directions: a refined theoretical understanding of implementability, and concrete advances in tool support for experimenting with increasingly expressive models.

6.1 Future Work

Future directions include extending the theoretical results beyond weak realizability toward a decidability result of *safe realizability* (therefore, including deadlock-freedom) for Global Types, building on the techniques developed here and extending an existing proof made by Lohrey, et al. [23]. On the practical side, a natural goal is to further enhance ReSCu to support these results, ultimately aiming for a complete algorithm to decide implementability for restricted classes of Global Types. This would enable systematic benchmarking against existing methods and real-world protocols.

References

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd international conference on Software engineering*, pages 304–313, 2000.

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623, 2003.

[3] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of msc graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[4] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In *International Conference on Coordination Languages and Models*, pages 86–106. Springer, 2020.

[5] Benedikt Bollig, Marie Fortin, and Paul Gastin. High-level message sequence charts: Satisfiability and realizability revisited. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 109–129. Springer, 2025.

[6] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[7] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):1–78, 2012.

[8] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8, 2012.

[9] Loïc Desgeorges and Loïc Germerie Guizouarn. Rsc to the rescu: Automated verification of systems of communicating automata. In *International Conference on Coordination Languages and Models*, pages 135–143. Springer, 2023.

[10] Loïc Desgeorges and Loïc Germerie Guizouarn. The original ReSCu repository. `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://src.koda.cnrs.fr/loic.germerie.guizouarn/rescu`, 2025. [Online; accessed 20-August-2025].

[11] Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. A partial order view of message-passing communication models. *Proceedings of the ACM on Programming Languages*, 7(POPL):1601–1627, 2023.

[12] Cinzia Di Giusto, Loïc Germerie Guizouarn, and Etienne Lozes. Multiparty half-duplex systems and synchronous communications. *Journal of Logical and Algebraic Methods in Programming*, 131:100843, 2023.

[13] Cinzia Di Giusto, Etienne Lozes, and Pascal Urso. Realisability and complementability of multiparty session types. *arXiv preprint arXiv:2507.17354*, 2025.

[14] Belot Florent. Drawing and Analyzing an MSC. `https://belotflorent.github.io/MSC-Tool-SWI-Prolog/`, 2024. [Online; accessed 15-August-2025].

[15] Loïc Germerie Guizouarn and Gabriele Genovese. The updated ReSCu repository. `https://github.com/gabrielegenovese/rescu`, 2025. [Online; accessed 20-August-2025].

[16] Loïc Germerie Guizouarn. *Communicating automata and quasi-synchronous communications*. PhD thesis, Université Côte d'Azur, 2023.

[17] Alexander Heußner, Tristan Le Gall, and Grégoire Sutre. Mcscm: a general framework for the verification of communicating machines. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 478–484. Springer, 2012.

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008.

[19] International Telecommunication Union. Z.120: Message Sequence Chart. Technical report, International Telecommunication Union, 1992.

[20] International Telecommunication Union. Z.120: Message Sequence Chart. Technical report, International Telecommunication Union, 1996.

[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. Communications of the ACM, 2019.

[22] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Complete multiparty session type projection with automata. In *International Conference on Computer Aided Verification*, pages 350–373. Springer, 2023.

[23] Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.

[24] Fabrizio Montesi. *Choreographic programming*. IT-Universitetet i København, 2014.

[25] Rémi Morin. Recognizable sets of message sequence charts. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 523–534. Springer, 2002.

[26] Felix Stutz. Asynchronous multiparty session type implementability is decidable–lessons learned from message sequence charts. *arXiv preprint arXiv:2302.11272*, 2023.

[27] Felix Stutz. *Implementability of Asynchronous Communication Protocols–The Power of Choice*. PhD thesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024.

[28] Felix Stutz and Emanuele D'Osualdo. An automata-theoretic basis for specification and type checking of multiparty protocols. In *European Symposium on Programming*, pages 314–346. Springer, 2025.

[29] Felix Stutz and Damien Zufferey. Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. *arXiv preprint arXiv:2208.05559*, 2022.

Appendices

ReSCu

Here are the details of the SCM grammar for the ReSCu's input.

```
prog          ::= <header> <aut_list> [<bad_confs>]
header        ::= scm <ident>:<channels> [<bags>] <parameters>
channels      ::= nb_channels = <int>;
bags          ::= //# bag_buffers = <int_list>
int_list      ::= <int>
                | <int_list>, <int>
parameters    ::= parameters = <param_list>
param_list    ::= <param>
                | <param> <param_list>
param         ::= {int | real} <ident>;
aut_list      ::= automaton <ident>:<initial>;<final>; <state_list>
initial       ::= initial : <int_list>;
final         ::= final : <int_list>;
state_list    ::= <state>
                | <state_list> <state>
state         ::= state <int> : <trans_list>
trans_list    ::= <transition>
                | <trans_list> <transition>
guard         ::= : when true | <nothing>
transition    ::= to <int> : when true , <int> <action> <ident>
action        ::= "!" | "?"
bad_confs     ::= bad_states: <bad_list>
bad_list      ::= (<bad_conf>)
                | <bad_list> (<bad_conf>)
bad_conf      ::= <bad_state>
                | <bad_state> with <bad_buffers>
bad_state     ::= automaton <ident>: in <int>: true [<bad_state>]
bad_buffers   ::= <regular_expression>
nothing       ::=
```

Listing 2: Simplified SCM grammar

Output for Example 3

```
This system is RSC.
There are some sink states:
Sink: Id=11 Configuration={{ F0:4; F1:3; P1:2; P2:2 }}
There are some deadlock states:
Deadlock: Id=4 Configuration={{ F0:2; F1:1; P1:1; P2:1 }}
Deadlock: Id=11 Configuration={{ F0:4; F1:3; P1:2; P2:2 }}
Deadlock: Id=8 Configuration={{ F0:4; F1:1; P1:1; P2:2 }}
Deadlock: Id=7 Configuration={{ F0:2; F1:3; P1:2; P2:1 }}
```

Listing 3: Output of Example 3

Output for Example 4

```
This system is RSC.
The system has the progress property.
There are some deadlock states:
Deadlock: Id=17 Configuration={{ A:1; B:4 }}
Deadlock: Id=15 Configuration={{ A:3; B:3 }}
```

Listing 4: Output of Example 4.

Figures for Example 3

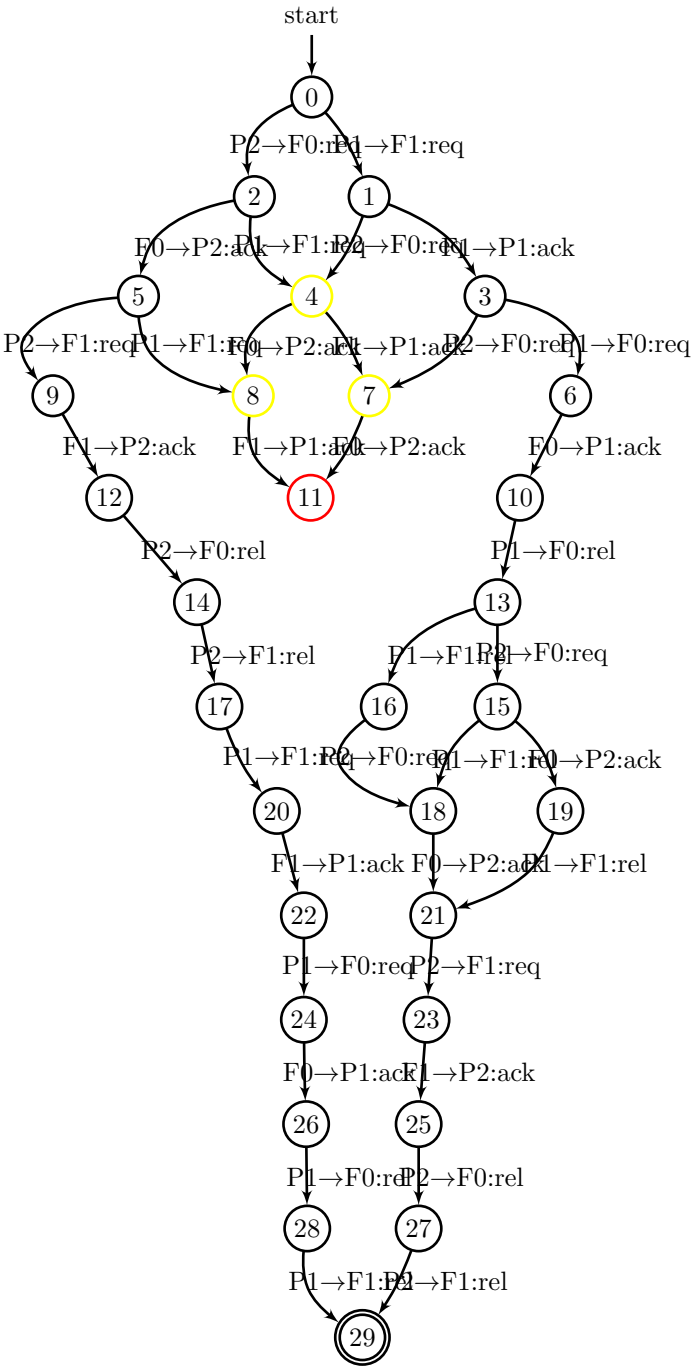


Figure 7: Synchronous Product of the Example 3.

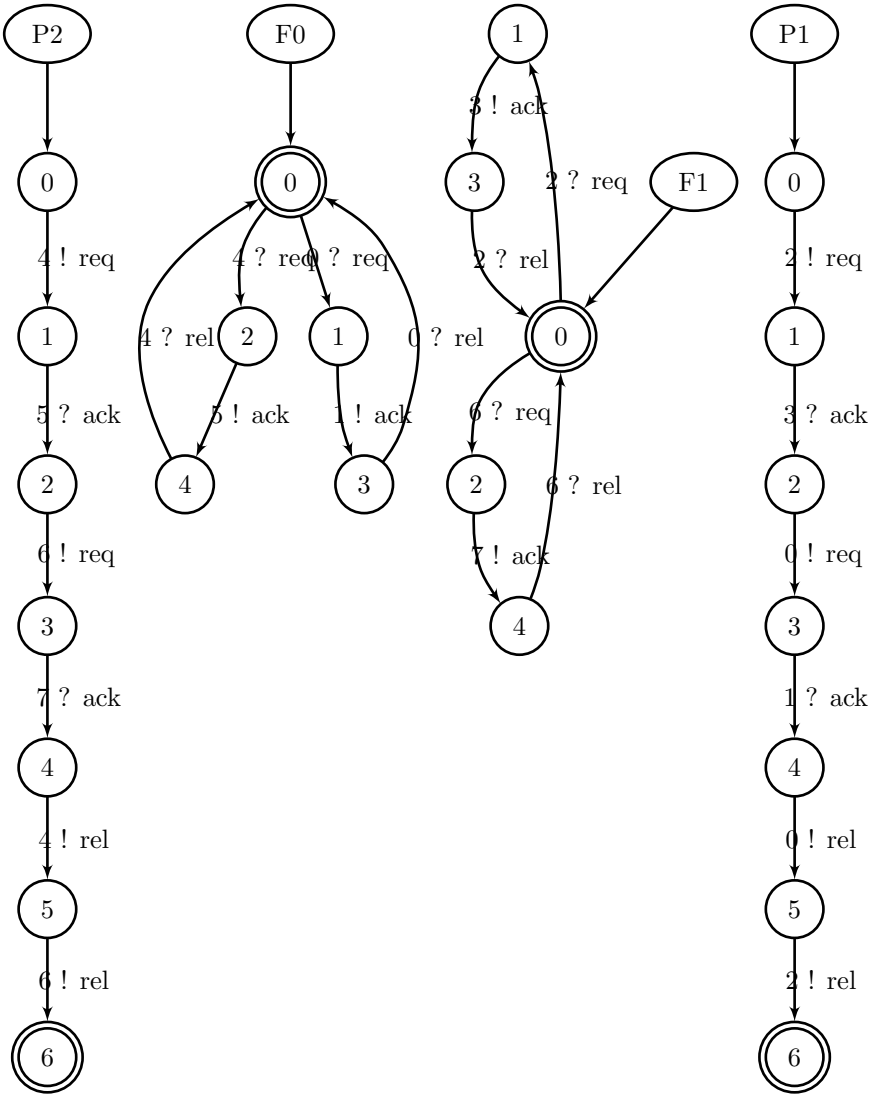


Figure 8: SCM automata representation of the Example 3.

Figures for Example 4

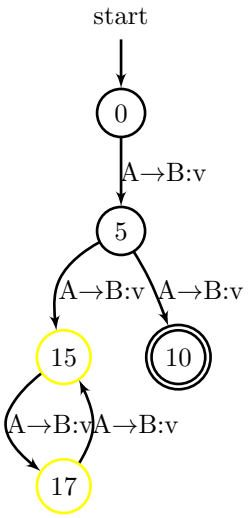


Figure 9: Synchronous Product of the Loop example

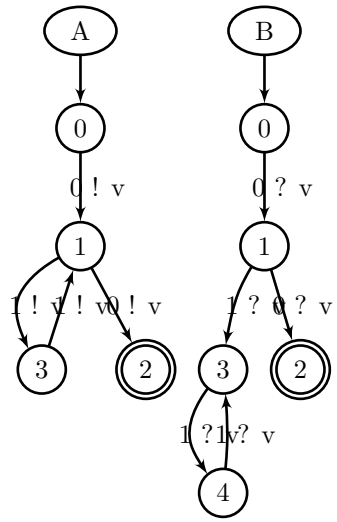


Figure 10: SCM automata representation of the Example 4.